Chapter 19

Ten Interesting Tools

In This Chapter

- Keeping track of application bugs
- Creating a safe place to test applications
- Getting your application placed on a user system
- Documenting your application
- Writing your application code
- Looking for application errors
- Working within an interactive environment
- Performing application testing
- Sorting the `import` statements in your application
- Keeping track of application versions

Python, like most other programming languages, has strong third-party support in the form of various tools. A *tool* is any utility that enhances the natural capabilities of Python when building an application. So, a debugger is considered a tool because it's a utility, but a library isn't. Libraries are instead used to create better applications. (You can see some of them listed in Chapter 20.)

Even making the distinction between a tool and something that isn't a tool, such as a library, doesn't reduce the list by much. Python enjoys access to a wealth of general-purpose and special tools of all sorts. In fact, the site at `https://wiki.python.org/moin/DevelopmentTools` breaks these tools down into the following 13 categories:

- AutomatedRefactoringTools
- BugTracking
- ConfigurationAndBuildTools
- DistributionUtilities
- DocumentationTools

- IntegratedDevelopmentEnvironments
- PythonDebuggers
- PythonEditors
- PythonShells
- SkeletonBuilderTools
- TestSoftware
- UsefulModules
- VersionControl

Interestingly enough, it's quite possible that the lists on the Python DevelopmentTools site aren't even complete. You can find Python tools listed in quite a few places online.

Given that a single chapter can't possibly cover all the tools out there, this chapter discusses a few of the more interesting tools — those that merit a little extra attention on your part. After you whet your appetite with this chapter, seeing what other sorts of tools you can find online is a good idea. You may find that the tool you thought you might have to create is already available, and in several different forms.

Tracking Bugs with Roundup Issue Tracker

You can use a number of bug-tracking sites with Python, such as the following: Github (`https://github.com/`); Google Code (`https://code.google.com/`); BitBucket (`https://bitbucket.org/`); and Launchpad (`https://launchpad.net/`). However, these public sites are generally not as convenient to use as your own specific, localized bug-tracking software. You can use a number of tracking systems on your local drive, but Roundup Issue Tracker (`http://roundup.sourceforge.net/`) is one of the better offerings. Roundup should work on any platform that supports Python, and it offers these basic features without any extra work:

- Bug tracking
- TODO list management

If you're willing to put a little more work into the installation, you can get additional features, and these additional features are what make the product special. However, to get them, you may need to install other products, such as a DataBase Management System (DBMS). The product instructions tell you what to install and which third-party products are compatible. After you make the additional installations, you get these upgraded features:

- Customer help-desk support with the following features:
  - Wizard for the phone answerers
  - Network links
  - System and development issue trackers
- Issue management for Internet Engineering Task Force (IETF) working groups
- Sales lead tracking
- Conference paper submission
- Double-blind referee management
- Blogging (extremely basic right now, but will become a stronger offering later)

Creating a Virtual Environment Using VirtualEnv

Reasons abound to create virtual environments, but the main reason for to do so with Python is to provide a safe and known testing environment. By using the same testing environment each time, you help ensure that the application has a stable environment until you have completed enough of it to test in a production-like environment. VirtualEnv (`https://pypi.python.org/pypi/virtualenv`) provides the means to create a virtual Python environment that you can use for the early testing process or to diagnose issues that could occur because of the environment. It's important to remember that there are at least three standard levels of testing that you need to perform:

- **Bug:** Checking for errors in your application
- **Performance:** Validating that your application meets speed, reliability, and security requirements
- **Usability:** Verifying that your application meets user needs and will react to user input in the way the user expects

Never test on a production server

A mistake that some developers make is to test their unreleased application on the production server where the user can easily get to it. Of the many reasons not to test your application on a production server, data loss has to be the most important. If you allow users to gain access to an unreleased version of your application that contains bugs that might corrupt the database or other data sources, the data could be lost or damaged permanently.

You also need to realize that you get only one chance to make a first impression. Many software projects fail because users don't use the end result. The application is complete, but no one uses it because of the perception that the application is flawed in some way. Users have only one goal in mind: to complete their tasks and then go home. When users see that an application is costing them time, they tend not to use it.

Unreleased applications can also have security holes that nefarious individuals will use to gain access to your network. It doesn't matter how well your security software works if you leave the door open for anyone to come in. After they have come in, getting rid of them is nearly impossible, and even if you do get rid of them, the damage to your data is already done. Recovery from security breaches is notoriously difficult — and sometimes impossible. In short, never test on your production server because the costs of doing so are simply too high.

Because of the manner in which most Python applications are used (see Chapter 18 for some ideas), you generally don't need to run them in a virtual environment after the application has gone to a production site. Most Python applications require access to the outside world, and the isolation of a virtual environment would prevent that access.

Installing Your Application Using PyInstaller

Users don't want to spend a lot of time installing your application, no matter how much it might help them in the end. Even if you can get the user to attempt an installation, less skilled users are likely to fail. In short, you need a surefire method of getting an application from your system to the user's system. Installers, such as PyInstaller (`http://www.pyinstaller.org/`), do just that. They make a nice package out of your application that the user can easily install.

Avoid the orphaned product

Some Python tools floating around the Internet are *orphaned*, which means that the developer is no longer actively supporting them. Developers still use the tool because they like the features it supports or how it works. However, doing so is always risky because you can't be sure that the tool will work with the latest version of Python. The best way to approach tools is to get tools that are fully supported by the vendor who created them.

If you absolutely must use an orphaned tool (such as when an orphaned tool is the only one available to perform the task), make sure that the tool still has good community support. The vendor may not be around any longer, but at least the community will provide a source of information when you need product support. Otherwise, you'll waste a lot of time trying to use an unsupported product that you might never get to work properly.

Fortunately, PyInstaller works on all the platforms that Python supports, so you need just the one tool to meet every installation need you have. In addition, you can get platform-specific support when needed. For example, when working on a Windows platform, you can create code-signed executables. Mac developers will appreciate that PyInstaller provides support for bundles. In many cases, avoiding the platform-specific features is best unless you really do need them. When you use a platform-specific feature, the installation will succeed only on the target platform.

A number of the installer tools that you find online are platform specific. For example, when you look at an installer that reportedly creates executables, you need to be careful that the executables aren't platform specific (or at least match the platform you want to use). It's important to get a product that will work everywhere it's needed so that you don't create an installation package that the user can't use. Having a language that works everywhere doesn't help when the installation package actually hinders installation.

Building Developer Documentation Using pdoc

Two kinds of documentation are associated with applications: user and developer. User documentation shows how to use the application, while developer documentation shows how the application works. A library requires only one sort of documentation, developer, while a desktop application may require only user documentation. A service might actually require

both kinds of documentation depending on who uses it and how the service is put together. The majority of your documentation is likely to affect developers, and pdoc (`https://github.com/BurntSushi/pdoc`) is a simple solution for creating it.

The pdoc utility relies on the documentation that you place in your code in the form of docstrings and comments. The output is in the form of a text file or an HTML document. You can also have pdoc run in a way that provides output through a web server so that people can see the documentation directly in a browser. This is actually a replacement for epydoc, which is no longer supported by its originator.

Developing Application Code Using Komodo Edit

Several chapters have discussed the issue of Interactive Development Environments (IDEs), but none have made a specific recommendation. The IDE you choose depends partly on your needs as a developer, your skill level, and the kinds of applications you want to create. Some IDEs are better than others when it comes to certain kinds of application development. One of the better general-purpose IDEs for novice developers is Komodo Edit (`http://komodoide.com/komodo-edit/`). You can obtain this IDE free, and it includes a wealth of features that will make your coding experience much better than what you'll get from IDLE. Here are some of those features:

- Support for multiple programming languages
- Automatic completion of keywords
- Indentation checking
- Project support so that applications are partially coded before you even begin
- Superior support

However, the thing that sets Komodo Edit apart from other IDEs is that it has an upgrade path. When you start to find that your needs are no longer met by Komodo Edit, you can upgrade to Komodo IDE (`http://komodoide.com/`), which includes a lot of professional level support features, such as code profiling (a feature that checks application speed) and a database explorer (to make working with databases easier).

Debugging Your Application Using pydbgr

A high-end IDE, such as Komodo IDE, comes with a complete debugger. Even Komodo Edit comes with a simple debugger. However, if you're using something smaller, less expensive, and less capable than a high-end IDE, you might not have a debugger at all. A *debugger* helps you locate errors in your application and fix them. The better your debugger, the less effort required to locate and fix the error. When your editor doesn't include a debugger, you need an external debugger such as pydbgr (`https://code.google.com/p/pydbgr/`).

A reasonably good debugger includes a number of standard features, such as code colorization (the use of color to indicate things like keywords). However, it also includes a number of nonstandard features that set it apart. Here are some of the standard and nonstandard features that make pydbgr a good choice when your editor doesn't come with a debugger:

- **Smart `eval`:** The `eval` command helps you see what will happen when you execute a line of code, before you actually execute it in the application. It helps you perform "what if" analysis to see what is going wrong with the application.
- **Out-of-process debugging:** Normally you have to debug applications that reside on the same machine. In fact, the debugger is part of the application's process, which means that the debugger can actually interfere with the debugging process. Using out-of-process debugging means that the debugger doesn't affect the application and you don't even have to run the application on the same machine as the debugger.
- **Thorough byte-code inspection:** Viewing how the code you write is turned into *byte code* (the code that the Python interpreter actually understands) can sometimes help you solve tough problems.
- **Event filtering and tracing:** As your application runs in the debugger, it generates events that help the debugger understand what is going on. For example, moving to the next line of code generates an event, returning from a function call generates another event, and so on. This feature makes it possible to control just how the debugger traces through an application and which events it reacts to.

Entering an Interactive Environment Using IPython

The Python shell works fine for many interactive tasks. You've used it extensively in this book. However, you may have already noted that the default shell has certain deficiencies (and if you haven't, you'll notice them as you work through more advanced examples). Of course, the biggest deficiency is that the Python shell is a pure text environment in which you must type commands to perform any given task. A more advanced shell, such as IPython (`http://ipython.org/`), can make the interactive environment friendlier by providing GUI features so that you don't have to remember the syntax for odd commands.

IPython is actually more than just a simple shell. It provides an environment in which you can interact with Python in new ways, such as by displaying graphics that show the result of formulas you create using Python. In addition, IPython is designed as a kind of front end that can accommodate other languages. The IPython application actually sends commands to the real shell in the background, so you can use shells from other languages such as Julia and Haskell. (Don't worry if you've never heard of these languages.)

One of the more exciting features of IPython is the ability to work in parallel computing environments. Normally a shell is single threaded, which means that you can't perform any sort of parallel computing. In fact, you can't even create a multithreaded environment. This feature alone makes IPython worthy of a trial.

Testing Python Applications Using PyUnit

At some point, you need to test your applications to ensure that they work as instructed. You can test them by entering in one command at a time and verifying the result, or you can automate the process. Obviously, the automated approach is better because you really do want to get home for dinner someday and manual testing is really, really slow (especially when you make mistakes, which are guaranteed to happen). Products such as PyUnit (`https://wiki.python.org/moin/PyUnit`) make unit testing (the testing of individual features) significantly easier.

The nice part of this product is that you actually create Python code to perform the testing. Your script is simply another, specialized, application that tests the main application for problems.

You may be thinking that the scripts, rather than your professionally written application, could be bug ridden. The testing script is designed to be extremely simple, which will keep scripting errors small and quite noticeable. Of course, errors can (and sometimes do) happen, so yes, when you can't find a problem with your application, you do need to check the script.

Tidying Your Code Using Isort

It may seem like an incredibly small thing, but code can get messy, especially if you don't place all your `import` statements at the top of the file in alphabetical order. In some situations, it becomes difficult, if not impossible, to figure out what's going on with your code when it isn't kept neat. The Isort utility (`http://timothycrosley.github.io/isort/`) performs the seemingly small task of sorting your `import` statements and ensuring that they all appear at the top of the source code file. This small step can have a significant effect on your ability to understand and modify the source code.

Just knowing which modules a particular module needs can be a help in locating potential problems. For example, if you somehow get an older version of a needed module on your system, knowing which modules the application needs can make the process of finding that module easier.

In addition, knowing which modules an application needs is important when it comes time to distribute your application to users. Knowing that the user has the correct modules available helps ensure that the application will run as anticipated.

Providing Version Control Using Mercurial

The applications you created while working through this book aren't very complex. In fact, after you finish this book and move on to more advanced training applications, you're unlikely to need version control. However, after you start working in an organizational development environment in which you create real applications that users need to have available at all times, version control becomes essential. *Version control* is simply the act of keeping

track of the changes that occur in an application between application releases to the production environment. When you say you're using MyApp 1.2, you're referring to version 1.2 of the MyApp application. Versioning lets everyone know which application release is being used when bug fixes and other kinds of support take place.

Numerous version control products are available for Python. One of the more interesting offerings is Mercurial (`http://mercurial.selenic.com/`). You can get a version of Mercurial for almost any platform that Python will run on, so you don't have to worry about changing products when you change platforms. (If your platform doesn't offer a binary, executable, release, you can always build one from the source code provided on the download site.)

Unlike a lot of the other offerings out there, Mercurial is free. Even if you find that you need a more advanced product later, you can gain useful experience by working with Mercurial on a project or two.

The act of storing each version of an application in a separate place so that changes can be undone or redone as needed is called *source code management.* For many people, source code management seems like a hard task. Because the Mercurial environment is quite forgiving, you can learn about source control management in a friendly environment. Being able to interact with any version of the source code for a particular application is essential when you need to go back and fix problems created by a new release.

The best part about Mercurial is that it provides a great online tutorial at `http://mercurial.selenic.com/wiki/Tutorial`. Following along on your own machine is the best way to learn about source control management, but even just reading the material is helpful. Of course, the first tutorial is all about getting a good installation of Mercurial. The tutorials then lead you through the process of creating a repository (a place where application versions are stored) and using the repository as you create your application code. By the time you finish the tutorials, you should have a great idea of how source control should work and why versioning is an important part of application development.